Designing a Secret Handshake: Authenticated Key Exchange as a Capability System

Dominic Tarr

June 14, 2015

Abstract

Capability Systems are a logical framework from which to compose secure systems, yet there is no widely implemented protocol for establishing secure two-way communication that also forms a capability system. We examin the ways in which other key exchange protocols fail to be suitable for a capability system, and then present a secure key exchange protocol designed with capability systems in mind. Using a preauthentication step, we authenticate the client before the server, but still acomplish mutual authentication within 4 passes. Included is a discussion of how the overall framework of capability systems guides the design process and simplifies many of the decisions involved in making a key exchange system.

1 Introduction

The development of key exchange algorithms marked the dawn of modern cryptography[5]. Their development was motivated by the desire for secure communications between two parties—yet designing a practical and secure protocol for exchanging a shared key between two authenticated parties is non-trivial[6].

Much of the research into key exchange has produced whole "families" of protocols [8]. Protocols currently in widespread use tend to be layered and configurable (TLS, SSH). This is not to the benefit of application developers—gaining a sufficiently nuanced understanding of such cryptosystems takes considerable study. Providing the developer with *more options*

is to provide them with *more opportunities* to add critical security flaws to their application. Thus, recent thought has steered towards providing simple constructions with security properties that are easy to understand and use safely [2]. We apply this philosophy to the design of an authenticated key exchange - a secret handshake. Since a key exchange can be designed with so many possible properties, we adopt the framework of capability systems[11] and allow that to drive the design. TahoeLAFS[12] is the inspiration for this decision. We find that no currently available handshake protocol adequately meets the needs of a capability system. Interestingly, a capability system demands a higher degree of privacy than is provided by other available protocols.

We will describe various key exchange protocols and examine why they are not suitable for a capabality system. We will discuss these protocols abstractly, giving sufficient details to demonstrate their cryptographic properties. In some cases, this means ignoring details required for their actual implementation, for example the handling of nonces. Finally, we will describe our capability based key exchange protocol.

2 Capability Systems

In a capability system, the right to access a resource is granted to the bearer of a *capability* (cap). Capability Based Security predates modern cryptography, but it's concepts map cleanly to the decentralized access control systems enabled by cryptography.

For example, possessing the key to decrypt a file means you have the *capability* to *read* that file. This can be implemented in a decentralized fashion: An encrypted file is widely distributed via a peer to peer network, but the read-cap is delegated only to a trusted few. Contrast this with an Access Control List (ACL), the file is kept on a server, and only those who can correctly authenticate can access the file: Like a bouncer at a nightclub, a trusted gatekeeper is required to administer the access list. A decentralized ACL is not possible. Unfortunately, we cannot cryptographically implement a fully featured object-capability system, we can implement the capabilities as keys[**?**] model.

Delegation is the act of granting an one of your access rights to another actor by sending them a capability. In a well behaved capability system, there is no way to gain a particular access right except through delegation.

Unintended Delegation. If you give your car keys to a valet to park your car, that is delegation, but if you leave your keys in your car and it is stolen–that is an *unintended delegation*. It is not the same as if someone forcibly gained entry to your car and hotwired it. If someone "steals" your car because you mistakenly gave them the keys, it is your own fault. If a system has unintended delegations at the protocol level, it is not a well behaved capability system.

Wildcard. In some card games there is a designated "wildcard" that can be played as any other card. A token is called a *wildcard* if it enables a set of rights that should have to be delegated individually. I know of no cryptosystem that is designed intentionally to contain wildcards. Presence of an unintended wildcard in a capability system should be considered a design failure.

Well Behaved. A capability system is *well behaved* if it supports delegation but lacks unintended delegations and wildcards.

3 Notation

$A \rightarrow B$ signifies a message passed from client to server, and $A \leftarrow B$ signifies a response. It's important to understand at what point a party becomes authenticated, and at what point that party knows it. If A or B is replaced with ? it means that party is not yet authenticated, to the knowledge of the receiver. If Alice receives a message $? \leftarrow B$ it means she now knows who Bob is, but Bob doesn't know who she is. A handshake is not complete until both parties know who they are talking to, and know the other party also knows. Thus a handshake does not end until two passes marked $A \rightarrow B$ and $A \leftarrow B$ are exchanged.

When a key pair is an argument to a function, it's represented as a subscript. When a key is sent in a message, only the public key is used, to be explicit I write that as A_p.

Long term keys for Alice and Bob are represented by A, B, and ephemeral keys are a, b. $a \cdot b$ denotes a shared key derived from a and b. Again, when this is a function argument it is shown as the subscript, $Box_{[a \cdot b]}(msg)$. Note that Box is authenticated encryption, it has the properties of a mac as well as of encryption. | denotes concatenation.

Finally, whenever a message is received it is immediately verified and if invalid the handshake is aborted.

4 Prior Art

4.1 Authenticated Key Exchange - STS, TLS, SSH

Here I will describe the Station To Station (STS) protocol[6]. This design forms the basis of most popular protocols (TLS, SSH) except without encrypting the keys or signatures (remember that Box is authenticated so $Box_k(msg)$ can be replaced with $mac_k(msg)$, and it will still constitute a proof of possession of the secret k.)

$$
\begin{aligned}
? &\rightarrow ? &&: a_p \\
? &\leftarrow B &&: b_p, Box_{[a \cdot b]}(B_p | Sign_B(a_p | b_p)) \\
A &\rightarrow B &&: Box_{[a \cdot b]}(A_p | Sign_A(a_p | b_p)) \\
A &\leftarrow B &&: Box_{[a \cdot b]}(okay)
\end{aligned}
$$

Alice sends a fresh ephemeral key to Bob, who creates one too, signs both keys and sends them back with his public key. Alice then boxes her public key and a signature to prove her identity, Bob then boxes a standard message to show his acceptance.

Neither party can be assured of the freshness of a message unless it is a cryptographic reply to a known fresh value, i.e. a value they freshly created. Hence, it is pointless for Alice to send her identity in the first pass. Bob cannot be sure it truly her until the third pass at the earliest.

However, Alice can know Bob's first message is fresh, so STS and many other protocols send the server authentication as the second pass. To resist an identity misbinding attack we require proof that the other party possesses the shared secret $a \cdot b$. Authenticated encryption, $Box_{[a \cdot b]}$, or a mac accomplishes that [8, section 3.1]

Often the description of a handshake protocol ends as soon as each party is authenticated, but before the client knows it is authenticated. If this is the case, the client could receive an authentication error (or dropped connection) at the application layer, which is awkward. Thus, they do not know they've been accepted until receiving the first encrypted message. For this reason I've represented STS as a 4 pass protocol, even though the original paper describes it as 3 pass protocol.

Is STS suitable for a capability system? No. The first thing STS does is authenticate the server, but the first question a capability system should ask is whether the client has the capability to this resource. Since Alice needs to know B_p to authenticate Bob *anyway*, it makes an excellent candidate for an access cap.

However, by authenticating Bob first, and sending his public key to an unauthenticated client, the public keys are delegated to anyone who initiates the protocol with Bob. Because of this unnecessary delegation, STS is not very useful as a capability system.

4.2 Encrypted Authenticated Key Exchange - CurveCP

CurveCP[1] implements authentication relying only on nacl's[2] *Box* primitive, which uses curve25519 keys. curve25519 keys are combined via scalar multiplication to produce shared keys, like Diffie Helman keys[5], not signing keys. CurveCP authenticates each party by showing they are able to produce the shared key, and it uses nested boxes to protect that authentication from eavesdroppers.

$$
\begin{aligned}
? &\rightarrow ? \ : a_p, Box_{[a \cdot B]}(okay) \\
? &\leftarrow B : Box_{[a \cdot B]}(b_p) \\
A &\rightarrow B : Box_{[a \cdot b]}(A_p || Box_{[A \cdot B]}(a_p)) \\
A &\leftarrow B : Box_{[a \cdot b]}(okay)
\end{aligned}
$$

Alice sends an ephemeral key, a_p to Bob, with a standard message encrypted to him - it's marked as $? \rightarrow ?$, though, because it may be a replay. Bob boxes his ephemeral key back to Alice's ephemeral key. This confirms Bob's identity to Alice–a man in the middle fails at this point. Alice then boxes her identity to Bob's ephemeral key, along with evidence that she is able to derive the key shared between Alice and Bob's long term identity (plus proof she did that after creating her ephemeral key). If Bob is satisified with this he responds with a standard message.

In CurveCP knowing Bob's public key functions as a capability to access Bob. Unfortunately, this protocol has two problems.

1. If Bob moves to a different address, a replay attack can confirm where,

without knowing his public key. Since the first packet is encrypted only Bob will be able to respond, so a response suggests it *is*. Having this information could encourage the replay attacker to look for other security weaknesses.

2. Worse, CurveCP contains a wildcard capability. By using curve25519 key sharing for authentication, CurveCP enables Key Compromise Impersonation (KCI) by someone who knows B_{secret} [4]. If Conrad gains B_{secret} he can impersonate *anyone* to Bob. To impersonate Alice, Condrad would connect to Bob, create an ephemeral key a, box it to Bob, and when Bob responds, create $Box_{[a \cdot b]}(A_p || Box_{[A \cdot B]}(a_p))$, except using (A_{public}, B_{secret}), as Bob would when opening the box, not (A_{secret}, B_{public}) as Alice would when sealing it. If Conrad possesses B_{secret} he can impersonate arbitary keys to Bob. Thus, B_{secret} is a wildcard capability, and CurveCP is not a well behaved capability system.

4.3 Deniable Authenticated Key Exchange - OTR, Noise, TextSecure

There is a class of key exchange protocols which make deniability a design goal[3, 9, 10]. The argument for this is that when you engage in casual communication you do not create evidence that you said what you did.

These protocols are unsuitable for a capability system for the same reason as STS—An unauthenticated client learns the server key in the second pass. This would be simple to fix, so whether a deniable key exchange can be a well behaved capability system deserves closer study. TextSecure[9] supercedes OTR[3], but uses a 3rd party introducer and is thus not directly analogous to a key exchange, so we will examine the noise[10] protocol instead.

Here we will extend the notation for shared keys to express keys shared between multiple pairs of keys. $a \cdot b | a \cdot B$ denotes the hash of $a \cdot b$ concatenated with $a \cdot B$. Only a party that can construct both the component keys can construct the composite key.

$$
\begin{aligned}
? &\rightarrow ? &&: a_p \\
? &\leftarrow B &&: b_p, Box_{[a\cdot b]}(B_p), Box_{[a\cdot b|a\cdot B]}(okay) \\
A &\rightarrow B &&: Box_{[a\cdot b]}(A_p), Box_{[a\cdot b|A\cdot b]}(okay) \\
A &\leftarrow B &&: Box_{[a\cdot b|a\cdot B|A\cdot b]}(a_p|b_p)
\end{aligned}
$$

Alice sends Bob her ephemeral key. Bob replies with his ephemeral key, a box containing his long term key, and a multikeyed box that as evidence he is Bob. Alice replies with a multikey box as she knows a_{secret} and A_{secret}., Bob then sends back a boxed message showing he was also able to derive that shared secret (like STS, it's really a 4 pass protocol)

Note that unlike CurveCP, a shared key is not derived between long term keys, but instead only between an ephemeral key and a long term key, thus B_{secret} is not a wildcard.

Even if Alice suspects that Conrad may have compromised her long term key, A, she trusts that he surely cannot know her ephemeral key, a. Without knowing a_{secret} Conrad cannot construct $a \cdot B$, unless it really *is* Bob. To provide the same assurance to Bob, they end up with the three way key $a \cdot b|a \cdot B|A \cdot b$, as in TextSecure[9]. This seems reasonable, except b_{secret} is now a wildcard! (though, since it is ephemeral, hopefully more difficult to gain)

1. The handshake is protected from eavesdroppers – but anyone who connects to the server will be sent the public key, so B_p is not useful as a capability.

2. KCI is still possible, but harder – To impersonate an arbitary key to Bob you have to know Bob's ephemeral *and* long term secret keys. This would be possible for anyone who had passive read access to Bob's memory – at first glance this may seem like a unlikely proposition, but in fact if Bob is running on a rented virtual machine that is precisely the situation he would be in. In a well behaved system, knowing Bob's private key *should* give you the ability to authenticate *as Bob*, but not as Alice.

Problem 2 would be avoided if the protocol was only ever run on physical hardware—but this is unreasonable. Since it still contains a wildcard this

protocol fails to form a well behaved capability system. Also, the property of deniability feels difficult to reason about. Will the higher level protocol introduce evidence of the communication? In any case, a secure channel is used for a lot more than casual social communications, and deniability does not appear to offer any special advantage to a capability system.

5 A New Design

If we take some ideas from the noise and TextSecure handshakes, but turn it around so that Alice authenticates first, then we get something that starts to look like a capability system.

If Alice *preauthenticates* Bob, then Bob can authenticate Alice using one more pass. With two initial passes to prevent replay attacks, we have a 4 pass protocol. This is no worse than the above, even though we do not authenticate anyone until the third pass.

To "preauthenticate", Alice sends a proof of both her identity, and her intention to connect to Bob. Preauthentication can be implemented with both encryption and signatures.

$$
\begin{aligned}
? &\rightarrow ? \ : a_p \\
? &\leftarrow ? \ : b_p \\
A &\rightarrow B : Box_{[a \cdot b | a \cdot B]}(A_p) \\
A &\leftarrow B : Box_{[a \cdot b | a \cdot B | A \cdot b]}(okay)
\end{aligned}
$$

Alice sends her ephemeral key to Bob, who responds with his. Then Alice boxes her long term key to Bob so that only he can open it. Bob shows his acceptance by boxing a standard message so that only Alice or Bob can read it.

Requiring Alice to authenticate first is unusual, but I think this is a fair deal. Bob has already put themself at a disadvantage by allowing himself to be publically addressable. It's only fair that Alice authenticates first. By encrypting her authentication she need not reveal her identity to anyone but the one true Bob. Likewise, if Bob chooses not to accept the call, then Alice won't be able to deduce whether or not it was really Bob. Maybe it was but

he did not wish to speak to her? Maybe it was just a wrong number? This protects Bob from harassment.

If Bob is unconcerned with the identity of his clients, he may allow anyone knowing B_p to authenticate. A client can remain anonymous by using a second ephemeral identity.

In this design Bob's public key forms as a capability, but it still has the wildcard (KCI) problem that noise and TextSecure have.

Key exchange is required for confidentiality and forward security, but signatures are required to avoid wildcards. With signatures, we'll have a truly well behaved capability system.

Since we will need both exchange and signing keys, an identity could be represented by a pair of signing and exchange keys. nacl uses ed25519 keys for signatures, and curve25519 keys for exchange. However, nacl also provides functions to convert signing to exchange keys, so an identity could be represented as a signing key, which would be converted to an exchange key when necessary.

$$
\begin{aligned}
? &\rightarrow ? : a_p \\
? &\leftarrow ? : b_p \\
H &= A_p | Sig_A(B_p | hash(a \cdot b)) \\
A &\rightarrow B : Box_{[a \cdot b | a \cdot B]}(H) \\
A &\leftarrow B : Box_{[a \cdot b | a \cdot B | A \cdot b]}(Sig_B(H))
\end{aligned}
$$

Alice and Bob exchange ephemeral keys, then Alice preauthenticates privately to Bob. Bob demonstrates his acceptance by privately signing her preauthentication. Note, that a_{secret} and b_{secret} are capabilites to decrypt the handshake, but since they are not wildcards we still have a well behaved capability system.

The design is getting much better. We resist eavesdropping, replay, man-in-the-middle, KCI attacks, and provide forward secrecy. There are no wildcards, nor unintended delegations. But there remain a few minor niggles to tidy up.

It is often helpful to have a protocol id and version associated with a handshake. Currently, we have no way to know if Alice and Bob are not speaking exactly the same protocol. Also, if either of the two initial passes

are tampered with, it would be undetected until the first authenticated pass is received. It also might be nice to prevent different applications built on this protocol from interfering with each other.

Most importantly, we must be careful to avoid creating a vulnerability to a Chosen Protocol Attack[7]. If the signing keys are also used elsewhere, it's possible that a signature from this protocol could be reused elsewhere, possibly creating an unintended delegation. All of these issues can be addressed via an application key (K) which forms a capability to the protocol. The ephemeral keys can be authenticated by using the K as the key to an *hmac*. By including K in each signature it is demonstrated that the signature belongs within this protocol, mitigating CPA. It is vital that if there are any other cases where a signing key is used, then a similar level of care is taken to prevent ambigious interpretations of signatures created.

K is a capability to access all actors that speak this protocol (or an application built on it). It should be updated if the implementation of the protocol changes sufficiently to be incompatible. If backwards compatibility is required, the new protocol version could be used on a new port, while the old version is being deprecated. For an openly specified application it may be known to the general public. For a private application it may be a closely guarded secret. An eavesdropper cannot extract K from the ciphertext, but it can confirm a correct guess at K. In the case where K is widely known, and an attacker is able to create a valid hmac, their interference will still be detected once the third message is received.

$$
\begin{aligned}
? \rightarrow ? &: a_p, hmac_K(a_p) \\
? \leftarrow ? &: b_p, hmac_{[K|a\cdot b]}(b_p) \\
H &= A_p|sign_A(K|B_p|hash(a \cdot b)) \\
A \rightarrow B &: box_{[K|a\cdot b|a\cdot B]}(H) \\
A \leftarrow B &: box_{[K|a\cdot b|a\cdot B|A\cdot b]}(sign_B(K|H|hash(a \cdot b)))
\end{aligned}
$$

The same as the previous, but at each pass, the shared key is extended as more public keys are learnt. Alice's authentication, $A_p|sign_A(K|B_p|hash(a \cdot b))$, proves that she possesses A, and that the proof is for this protocol (via K) and this handshake (via $hash(a \cdot b)$). Incase Bob does not have alice's key yet, we delegate it to him, (obviously we want Bob to have that capability,

because otherwise Alice would not have connected to him)

For Bob to authenticate back to Alice, he could just sign the proof Alice sent him, and send it back. H is already cryptographically linked to the preceding passes. On the other hand, it is easier to persuade ourselves that signing K mitigates CPA[7] than that A_p is never a special value in another protocol.

Alice and Bob can now use their shared secret, $K|a \cdot b|a \cdot B|A \cdot b$, with a bulk encryption protocol to secure a two-way communication channel.

6 Future Work

The latency induced by a 4 pass protocol may be prohibitive for some applications. A mechanism to prearrange a single-use key for the next session may enable a two pass protocol, at least once a given pair of actors have established contact.

Some readers will be wondering how Alice is to learn Bob's public key? The conceptual framework of capability systems has a simple answer: someone delegates it to her. In practice, there is a large design space in how and why someone may wish to do so. Briefly, there could be a centralized registry, a DHT, a gossip network, or access caps could be configured in files. Various combinations of the above are likely, or different systems entirely.

7 Conclusion

I have described a highly private 4 pass handshake protocol that is suitable for capability systems. It does not suffer from replay, eavesdropping, man in the middle, or key compromise impersonation. Capability Systems provide a conceptual framework for security based not on distributing access, but by restricting it. In a cryptographic capability system like this one, public and private keys are simply *access rights*. This allows us to think otherwise unthinkable thoughts, such as the notion of a *secret public key* or a *shared private key*. With concepts like these, we could create dynamic layers of access and restrictions. Since *Secret Handshake* is otherwise unopinionated, and secure two-way communication is fundamental, if it was combined with a suitable streaming bulk encryption protocol, it could form a powerful building block for decentralized access control systems.

References

[1] Daniel Bernstein. Curvecp: Usable security for the internet. http://curvecp.org/index.html, Feb 2011. Accessed: 2015-06-10.

[2] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2Nd International Conference on Cryptology and Information Security in Latin America*, LATINCRYPT'12, pages 159–176, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[4] CodesInChaos. Curvecp review - part 1 crypto. https://codesinchaos.wordpress.com/2012/09/09/curvecp-1/, Sept 2012.

[5] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[6] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges, 1992.

[7] John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In *In Proc. 1997 Security Protocols Workshop*, pages 91–104. Springer-Verlag, 1997.

[8] Hugo Krawczyk. Sigma: the sign-and-mac approach to authenticated diffie-hellman. In *SIGMA, full version. http://www.ee.technion.ac.il/ hugo/sigma.html*, 2003.

[9] Moxie Marlinspike. Simplifying otr deniability. https://whispersystems.org/blog/simplifying-otr-deniability/, July 2013.

[10] Trevor Perrin. noise. https://github.com/trevp/noise/blob/master/noise.md, Aug 2014.

[11] Tristan Slominski. Towards a universal implementation of unforgeable actor addresses. http://www.dalnefre.com/wp/2013/10/towards-a-universal-implementation-of-unforgeable-actor-addresses/, Oct 2013. Accessed: 2015-06-10.

[12] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe the least-authority filesystem. Cryptology ePrint Archive, Report 2012/524, 2012.